به نام خدا

تمرین 5 طراحی زبان

سرکار جناب آقای دکتر ایزدی

سارا آذرنوش

98170668

(1

(value-of exp1 e s0) = (l, s1)

(value-of exp2 e s1) = (val, s2)

(value-of (setref-exp exp1 exp2) e s0) = (val, [l=val]s2)

(value-of exp1 e s0) = (l, s1)

(value-of (deref-exp exp1) e s0) = (l, s1)

(value-of exp1 e s0) = (val,s1)

(value-of (newref-exp exp1) e s0) = (ref-val, [l=val]s1)

```
#lang racket
(define value-of
  (lambda (exp env)
    (cases expression exp
        (newref-exp (exp1)
                (let ((v1 (value-of exp1 env)))
                  (ref-val (newref v1))))
        (deref-exp (exp1)
               (let ((v1 (value-of exp1 env)))
                 (let ((ref1 (expval->ref v1)))
                   (deref ref1))))
        (setref-exp (exp1 exp2)
                (let ((ref (expval->ref (value-of exp1 env))))
                        (let ((val2 (value-of exp2 env)))
                          (begin
                            (setref! ref val2)
                            val2))))

     )))

(define the-grammar
  '((program (expression) a-program)
    (expression("newref" "(" expression ")")newref-exp)
    (expression ("deref" "(" expression ")")deref-exp)
    (expression("setref" "(" expression "," expression ")")setref-exp)
    ))

(define-datatype expval expval?
  (ref-val
   (ref reference?))

(define-datatype expression expression?
  (newref-exp(exp1 expression?))
  (deref-exp(exp1 expression?))
  (setref-exp
   (exp1 expression?)
   (exp2 expression?)))

(define expval->ref
  (lambda (val)
  (cases expval val
      (ref-val (ref) ref)
      (else (report-expval-extractor-error 'reference val)))))
```

(2

Begin-end

(value-of exp0 e s0) = (val0, s1)

(value-of exp1 e s1) = (val1, s2)

...

(value-of expn e sn) = (valn, sn+1)

(value-of (begin-exp (exp0, exp1, ..., expn)), e, s0) = (valn, sn+1)

list

(value-of exp0 e s0) = (val0, s1)

(value-of exp1 e s1) = (val1, s2)

...

(value-of expn e sn) = (valn, sn+1)

(value-of (list-exp (exp0, exp1, ..., expn)), e, s0)
= ((val0, val1, ..., valn), sn+1)

New-list

(value-of exp1 e s0) = (val,s1)

(value-of (newlist-exp exp1) e s0) = (num-val, s1)

Get-list

(value-of exp1 e s0) = (val0, s1)

(value-of exp2 e s1) = (val1, s2)

(value-of (getlist-exp exp1 exp2) e s0) = ((val0, val1), s2)

Set-list

(value-of exp1 e s0) = (val0, s1)
(value-of exp2 e s1) = (val1, s2)
(value-of exp3 e s2) = (val2, s3)

(value-of (setlist-exp exp1 exp2 exp3) e s0) = ((val0, val1,val2), s2)

(3

```
#lang racket
(define value-of
 (lambda (exp env)
  (cases expression exp
      (begin-exp (exp1 exps)
           (letrec
             ((value-of-begins
              (lambda (e1 es)
               (let ((v1 (value-of e1 env)))
                (if (null? es)
                  v1
                  (value-of-begins (car es) (cdr es)))))))
            (value-of-begins exp1 exps)))

      (setlist-exp (exps1 exps2 exps3)//list index value
           (if (null? exps)
             (list-val '())
             (list-val
              (append (exp3 (list (value-of (car exps1) env))
                 (list (value-of (list-exp (cdr exps1))env))))))
      (newlist-exp (exp1)
           (let ((val1 (value-of exp1 env)))
             (let ((num1 (expval->num val1))))))
      (get-list-exp (exp exp2)
          (let ([l (value-of exp env store)])
          (let ([i (value-of exp2 env (car l))])
          (list-ref (expval->exp (cadr l))(expval->int (cadr i))))))
   )))


(define the-grammar
 '((program (expression) a-program)
  (expression ("begin" expression (arbno ";" expression) "end")begin-exp)
  [expression ("newlist" "(" expression ")") newlist-exp]
  (expression ("getlist" "(" expression "," expression ")") getlist-exp)
  (expression ("setlist" "(" expression "," expression "," expression ")") setlist-exp)
  ))


(define-datatype expression expression?
 (begin-exp(exp1 expression?)(exps (list-of expression?)))
 (newlist-exp(size number?))
 (getlist-exp(exp1 expression?)(exp2 expression?))
 (setlist-exp(exp1 expression?)(exp2 expression?) (exp3 expression?))
 )
```

```
(define expval->list
  (lambda (val)
    (cases expval val
           (bool-val (lst) lst)
           (else (report-expval-extractor-error 'list val)))))
```

(4

```
#lang racket
(define the-grammar
  '((program (expression) a-program)
    (expression("(" expression expression ")")call-exp)
    (expression("letrec"(arbno identifier "(" identifier ")" "=" expression)"in" expression)letrec-exp)
     ))

  (define value-of
  (lambda (exp env)
    (cases expression exp
           (call-exp (rator rand)
                     (let ((proc (expval->proc (value-of rator env)))
                           (arg (value-of rand env)))
                       (apply-procedure proc arg)))
           (letrec-exp (proc-names bound-vars proc-bodies letrec-body)
                       (value-of letrec-body
                                 (extend-env-rec*
                                  proc-names bound-vars proc-bodies env)))
    )))


 (define-datatype environment environment?
  (empty-env)
  (extend-env(saved-var symbol?) (saved-ref reference?) (saved-env environment?))
  (extend-env-rec*
   (p-names (list-of identifier?))(b-vars (list-of identifier?))(bodies (list-of expression?))(saved-env
environment?)))
```

```
(define-datatype expression expression?
  (call-exp
   (rator expression?)(rand expression?))
  (letrec-exp
   (proc-names (list-of identifier?))(bound-vars (list-of identifier?))(proc-bodies (list-of
expression?))(letrec-body expression?))
)
```

5)

6)

توابع bool و Zero برای بول استفاده میشوند.

```
#lang racket
(define type-of
  (lambda (exp tenv)
    (cases expression exp
   [zero?-exp (exp1) (let ([ty1 (type-of exp1 tenv)])
             (check-equal-type! ty1 (int-type) exp1)
             (bool-type))]
   [if-exp (exp1 exp2 exp3) (let ([ty1 (type-of exp1 tenv)]
                   [ty2 (type-of exp2 tenv)]
                   [ty3 (type-of exp3 tenv)])
                 (check-equal-type! ty1 (bool-type) exp1)
                 (check-equal-type! ty2 ty3 exp)
                 ty2)]
   )))

(define the-grammar
 '([program (expression) a-program]
  [expression ("zero?" "(" expression ")") zero?-exp]
  [expression ("if" expression "then" expression "else" expression) if-exp]
  ))

(define check-equal-type!
 (lambda (ty1 ty2 exp)
  (when (not (equal? ty1 ty2))
   (report-unequal-types ty1 ty2 exp))))

(define expval->bool
 (lambda (v)
  (cases expval v
   [bool-val (bool) bool]
   [num-val (num) (if (equal? num 0) #f #t])
```

```
      [else (expval-extractor-error 'bool v)]))))


(define value-of
  (lambda (exp env)
    (cases expression exp
      [zero?-exp (exp1) (let ([val1 (expval->num (value-of exp1 env))])
                          (if (zero? val1)
                              (bool-val #f)
                              (bool-val #t)))]
      [if-exp (exp0 exp1 exp2) (if (expval->bool (value-of exp0 env))
                                   (value-of exp1 env)
                                   (value-of exp2 env))])))
```

7)

1.

```
letrec ? even (x : ?) 2 = if zero?(x) then 1 else (odd -(x, 1)) 3
 ? odd (x : ?) 4 = if zero?(x) then 0 else (even -(x, 1)) 5 in (odd 13)
```

با توجه به گفته کلاس کل آن int است و هر کدام Int -> int است.

2.

```
let p = zero?(1) in if p then 88 else 99
bool -> int
```

3.

```
let f = proc (z) z in proc (x) -((f x), 1)
 ((t -> int) -> (t -> int))
```

8)

توابع bool و Zero برای بول استفاده میشوند.

```
#lang racket
(define type-of
 (lambda (exp tenv subst)
  (cases expression exp
   [zero?-exp (exp1) (cases answer (type-of exp1 tenv subst)
             [an-answer (type1 subst1) (let ([subst2 (unifier type1 (int-type) subst1 exp)])
                           (an-answer (bool-type) subst2))])]
   [if-exp (exp1 exp2 exp3)
       (cases answer (type-of exp1 tenv subst)
        [an-answer (ty1 subst) (let ([subst (unifier ty1 (bool-type) subst exp1)])
                    (cases answer (type-of exp2 tenv subst)
                     [an-answer (ty2 subst) (cases answer (type-of exp3 tenv subst)
                                  [an-answer (ty3 subst) (let ([subst (unifier ty2
                                                        ty3
                                                        subst
                                                        exp)])
                                              (an-answer ty2
                                                    subst))])]))])]
   )))
(define value-of
 (lambda (exp env)
  (cases expression exp
   [zero?-exp (exp1) (let ([val1 (expval->num (value-of exp1 env))])
             (if (zero? val1)
               (bool-val #f)
               (bool-val #t)))]
   [if-exp (exp0 exp1 exp2) (if (expval->bool (value-of exp0 env))
                 (value-of exp1 env)
                 (value-of exp2 env))]
   )))

(define expval->bool
 (lambda (v)
  (cases expval v
   [bool-val (bool) bool]
   [num-val (num) (if (equal? num 0) #f #t)]
   [else (expval-extractor-error 'bool v)])))

(define the-grammar
 '([program (expression) a-program]
  [expression ("zero?" "(" expression ")") zero?-exp]
  [expression ("if" expression "then" expression "else" expression) if-exp])
```